Chapter 6 • Allen & Kennedy, *Optimizing Compilers for Modern Architectures*

Creating Coarse-Grained Parallelism

Introduction

- Chapter 6: Focus on parallelism for SMPs
  - Contrast with Chapter 5 (vector and superscalar processors)
  - Focus on parallelizing *outer loops*
    - Often contain large blocks of work in each iteration
  - Thread creation, barrier synchronization expensive
    - Tradeoff: synchronization overhead vs. parallelism/load balance
- Transformations that uncover coarse-grained parallelism
  1. Define or review each transformation
  2. Contrast with use in Chapter 5 (if applicable)
  3. Describe effect on dependences
  4. Discuss when it can/should be applied

Overview

- Transformations on Single Loops
    - Privatization, Alignment, Code Replication, Loop Distribution & Fusion
- Transformations on Perfect Loop Nests
    - Loop Interchange, Loop Skewing
- Transformations on Imperfectly Nested Loops
    - Multilevel Loop Fusion

I. Single-Loop Methods

Privatization
Loop Distribution
Alignment
Code Replication
Loop Fusion

Focus on…

(1) Parallelizing sequential loops
(2) Increasing granularity of parallel loops

Scalar Privatization (1/4)

The Transformation

- Make a variable used only within an iteration private

```
DO I = 1, N
    T = A(I)
    A(I) = B(I)
    B(I) = T
ENDDO
```

```
PARALLEL DO I = 1, N
    PRIVATE t
    t = A(I)
    A(I) = B(I)
    B(I) = t
END PARALLEL DO
```

Scalar Privatization (2/4)

Comparison with Chapter 5

- Similar to scalar expansion
  - Also useful in parallelization (p. 243)
- But privatization better for SMPs
- Like scalar expansion, not cost-free

```
DO I = 1, N
      T$(I) = A(I)
      A(I) = B(I)
      B(I) = T$(I)
ENDDO
T = T$(N)
```

```
DO I = 1, N
    T = A(I)
    A(I) = B(I)
    B(I) = T
ENDDO
```

```
PARALLEL DO I = 1, N
    PRIVATE t
    t = A(I)
    A(I) = B(I)
    B(I) = t
END PARALLEL DO
```

Scalar Privatization (3/4)

Effect on Dependences

- Eliminates loop-carried and loop-independent dep's associated with a scalar
    - Like scalar expansion
    - Makes loop parallelizable

```
DO I = 1, N
    T = A(I)
    A(I) = B(I)
    B(I) = T
ENDDO
```

```
PARALLEL DO I = 1, N
    PRIVATE t
    t = A(I)
    A(I) = B(I)
    B(I) = t
END PARALLEL DO
```

Scalar Privatization (4/4)

When to Privatize a Scalar in a Loop Body

- When all dep's carried by a loop involve a privatizable variable
  - *Privatizable:* Every use follows a definition (in the loop body)
  - Equivalently, no upwards-exposed uses in the loop body
  - Determine privatizability through data flow analysis (or SSA form – p.242)
  - If cannot privatize, try scalar expansion (p. 243)

```
DO I = 1, N
    T = A(I)
    A(I) = B(I)
    B(I) = T
ENDDO
```

Array Privatization

- Make an array used only within an iteration private

```
DO I = 1, 100
  T(1) = X
  DO J = 2, N
    T(J) = T(J-1)+B(I,J)
    A(I,J) = T(J)
  ENDDO
ENDDO
```

```
PARALLEL DO I = 1, 100
  PRIVATE t(N)
  t(1) = X
  DO J = 2, N
    t(J) = t(J-1) + B(I,J)
    A(I,J) = t(J)
  ENDDO
  IF (I==100) T(1:N) = t(1:N)
ENDDO
```

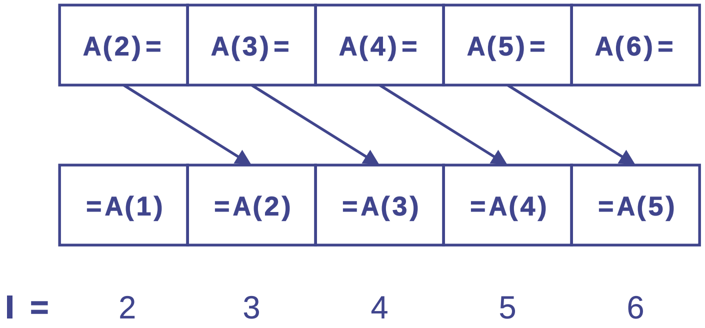
- Overview of finding privatizable arrays: p. 244

Loop Alignment (1/4)

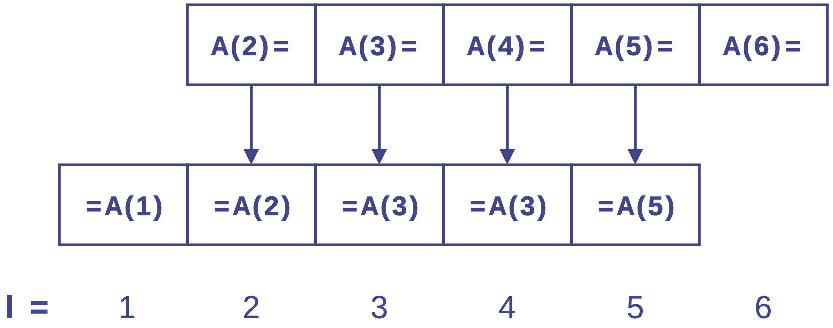
Effect on Dependences

- **Problem:** Source computed on iteration prior to sink

```
DO I = 2, N
  A(I) = B(I)+C(I)
  D(I) = A(I-1)*2.0
ENDDO
```



- **Solution:** Compute sources and sinks on same iteration



Loop Alignment (2/4)

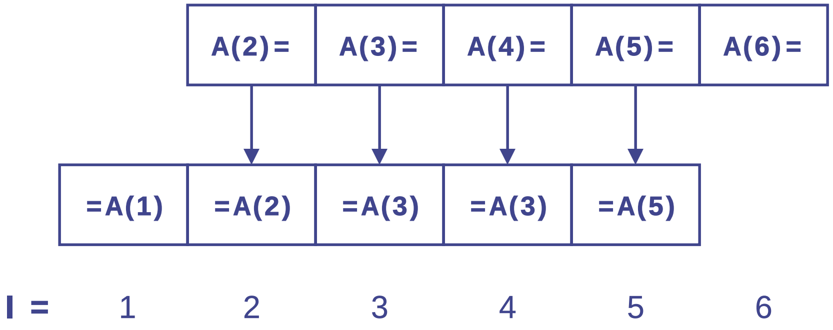
The Transformation

- Naive implementation

```
DO I = 2, N
  A(I) = B(I)+C(I)
  D(I) = A(I-1)*2.0
ENDDO
```

```
DO i = 1, N
  IF (i>1) A(i) = B(i)+C(i)
  IF (i<N) D(i+1) = A(i)*2.0
ENDDO
```

- Overhead due to extra iteration and conditional tests can be reduced…



I. Single Loop Methods

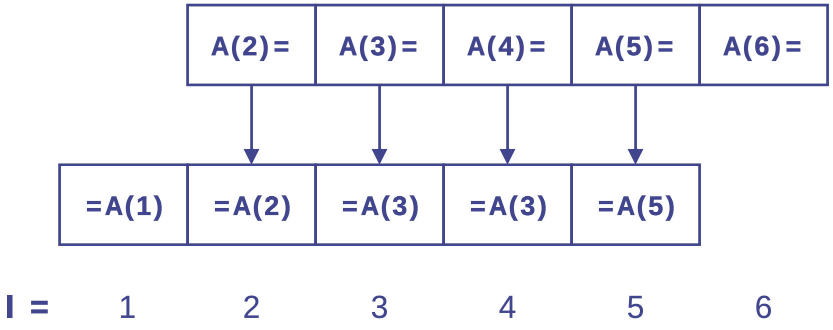
Loop Alignment (3/4)

The Transformation

- Improved implementation (Eliminates extra iteration & conditionals)

```
DO I = 2, N
  A(I) = B(I)+C(I)
  D(I) = A(I-1)*2.0
ENDDO
```

```
D(2) = A(1)*2.0
DO i = 2, N-1
  A(i) = B(i)+C(i)
  D(i+1) = A(i)*2.0
ENDDO
A(N) = B(N)+C(N)
```



A(2)= A(3)= A(4)= A(5)= A(6)=

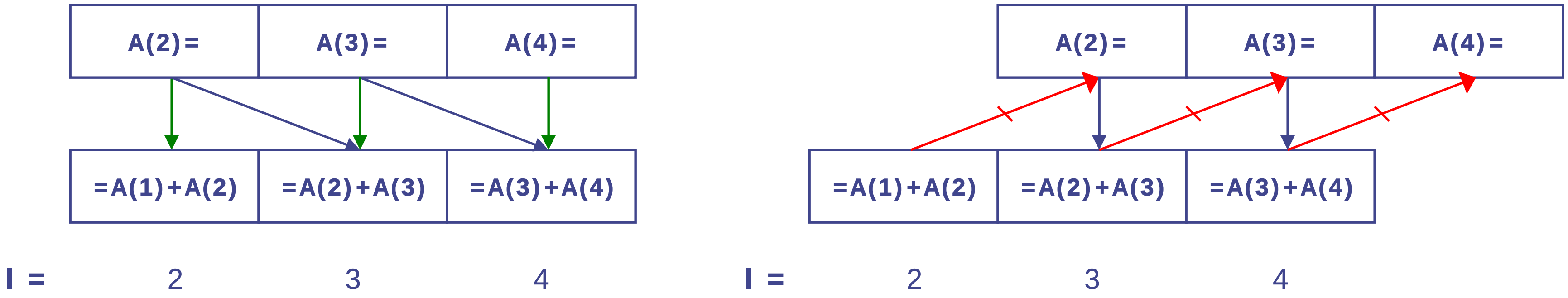
=A(1) =A(2) =A(3) =A(3) =A(5)

I = 1 2 3 4 5 6

Loop Alignment (4/4)

When NOT to Apply

- Alignment cannot eliminate a carried dependence in a recurrence (p. 248)
- Also **alignment conflicts:** two dependences can't be simultaneously aligned
  - Example:



A(2)= A(3)= A(4)=

=A(1)+A(2) =A(2)+A(3) =A(3)+A(4)

I = 2 3 4

A(2)= A(3)= A(4)=

=A(1)+A(2) =A(2)+A(3) =A(3)+A(4)

I = 2 3 4

When TO Apply

- Applied along with *Code Replication,* so let's discuss that first…

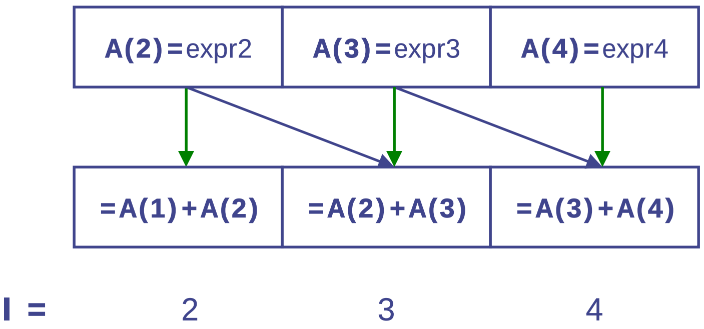
Code Replication (1/2)

Effect on Dependences

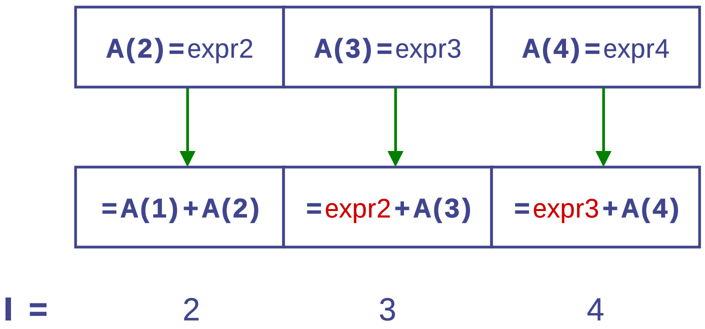
- Want to eliminate alignment conflicts by eliminating loop-carried deps

The Transformation

- Replace the code at the sink of a loop-carried dependence with the expression computed at the source



I =	2	3	4
	A(2)=expr2	**A(3)=**expr3	**A(4)=**expr4
	=A(1)+A(2)	**=A(2)+A(3)**	**=A(3)+A(4)**



I =	2	3	4
	A(2)=expr2	**A(3)=**expr3	**A(4)=**expr4
	=A(1)+A(2)	**=**expr2**+A(3)**	**=**expr3**+A(4)**

Code Replication (2/2)

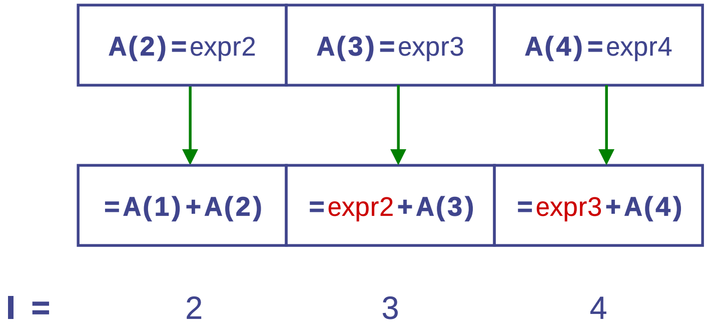
```
DO I = 1, N
  A(I+1) = B(I)+C
  X(I)   = A(I+1)+A(I)
ENDDO
```

```
DO I = 1, N
  A(I+1) = B(I)+C

  IF (I==1) THEN
    t = A(I)
  ELSE
    t = B(I-1) + C
  END IF

  X(I) = A(I+1)+t
ENDDO
```

- **The Transformation**



Alignment & Replication

- **Effect on Dependences**
  - Both eliminate loop-carried dependences
- **When to Align Loops and/or Replicate Code**
  - Obviously, replication has a higher cost; alignment is preferable
  - “Alignment, replication, and statement reordering are sufficient to eliminate all carried dependences in a single loop that contains no recurrence and in which the distance of each dependence is a constant independent of the loop index.” (Theorem 6.2)
    - Proved constructively
    - read §6.2.4 for full detail

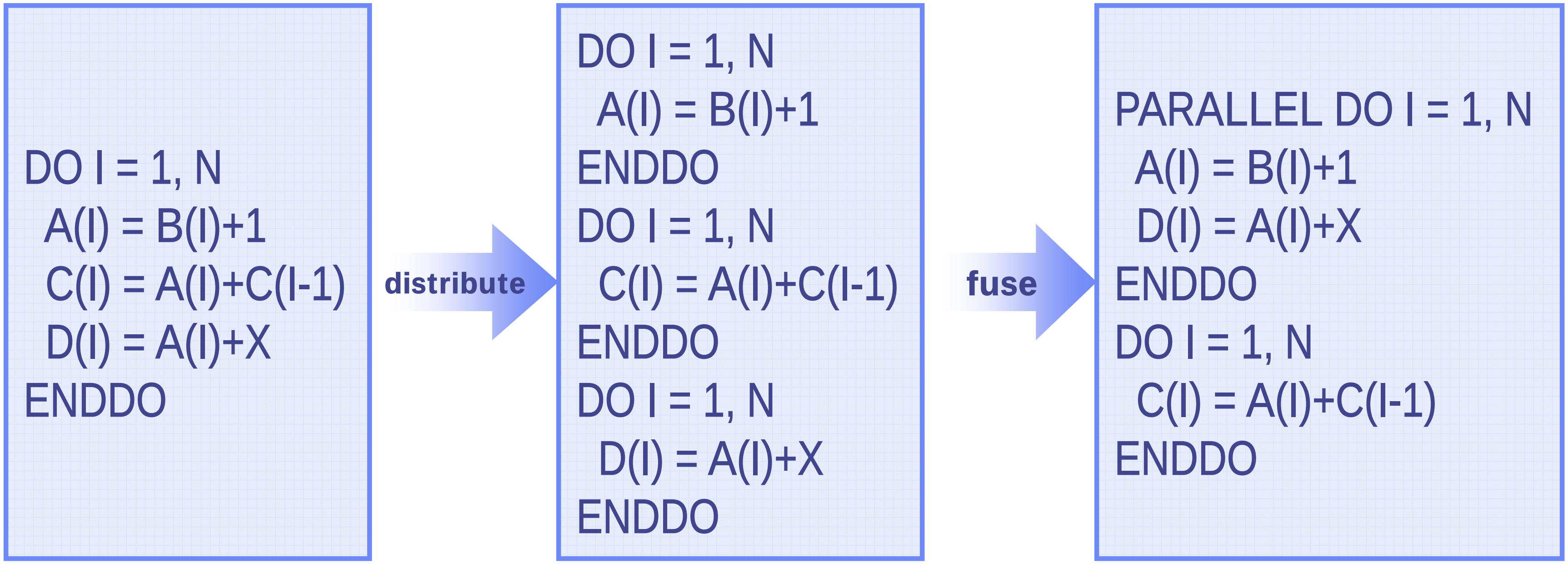
Loop Distribution ("Loop Fission")

- Also eliminates carried dependences
  - Smaller loop bodies ⇒ Decreased granularity
    - This was good in Chapter 5 (vectorization); bad for SMPs
  - Converts to loop-independent deps *between loops*
  - ⇒ Implicit barrier between loops ⇒ Sync overhead
  - ∴ Try privatization, alignment, and replication first
- Use to separate potentially-parallel code from necessarily-sequential code in a loop
  - Can recover granularity:
    - Use maximal loop distribution, then
    - Recombine ("fuse") loops…

Loop Fusion (1/6)

The Transformation

- Combine 2+ distinct loops into a single loop



```
DO I = 1, N
  A(I) = B(I)+1
  C(I) = A(I)+C(I-1)
  D(I) = A(I)+X
ENDDO
```

```
DO I = 1, N
  A(I) = B(I)+1
ENDDO
DO I = 1, N
  C(I) = A(I)+C(I-1)
ENDDO
DO I = 1, N
  D(I) = A(I)+X
ENDDO
```

```
PARALLEL DO I = 1, N
  A(I) = B(I)+1
  D(I) = A(I)+X
ENDDO
DO I = 1, N
  C(I) = A(I)+C(I-1)
ENDDO
```

Loop Fusion (2/6)

When to Fuse Loops: Safety Constraints

- **1. No fusion-preventing dependences**
  - **Def. 6.3:** A loop-independent dependence between statements in two different loops is *fusion preventing* if fusing the two loops causes the dependence to be carried by the combined loop in the reverse direction
  - Note that distributed loops can always be fused back together

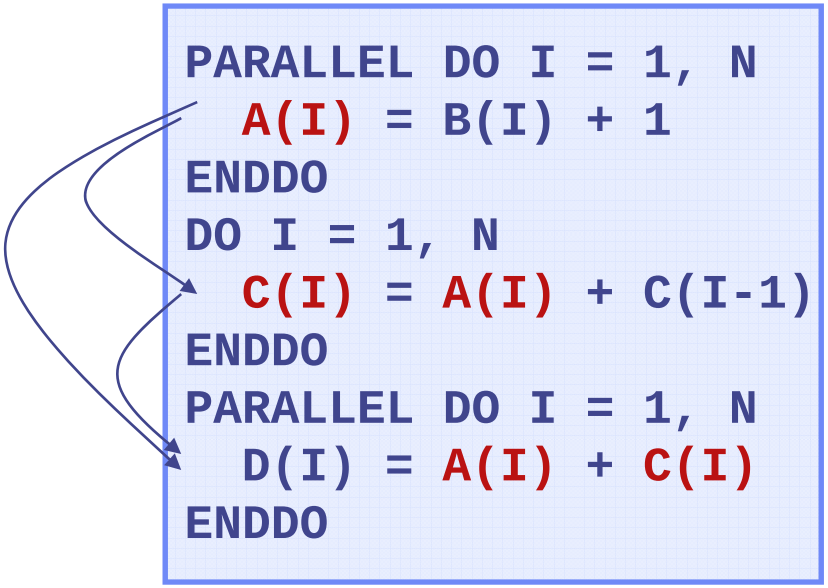
```
DO I = 1, N
  A(I) = B(I) + C
ENDDO
DO I = 1, N
  D(I) = A(I+1) + E
ENDDO
```

→

```
DO I = 1, N
  A(I) = B(I) + C
  D(I) = A(I+1) + E
ENDDO
```

Loop Fusion (3/6)

- **When to Fuse Loops: Safety Constraints**
  - **2. No invalid reordering**
    - Two loops cannot be fused if there is a path of loop-independent dependences between them that contains a loop or statement that is not being fused with them



```
PARALLEL DO I = 1, N
  A(I) = B(I) + 1
ENDDO
DO I = 1, N
  C(I) = A(I) + C(I-1)
ENDDO
PARALLEL DO I = 1, N
  D(I) = A(I) + C(I)
ENDDO
```

Loop Fusion (4/6)

- **When to Fuse Loops: Profitability Constraints**
  - **3. Separate sequential loops**
    - Do not fuse sequential loops with parallel loops:
      The result would be a sequential loop

Loop Fusion (5/6)

- **When to Fuse Loops: Profitability Constraints**
  - **4. No parallelism-inhibiting dependences**
    - Do not fuse two loops if a fusion would cause a dependence between the two original loops to be carried by the combined loop

```
DO I = 1, N
  A(I+1) = B(I) + C
ENDDO
DO I = 1, N
  D(I) = A(I) + E
ENDDO
```

```
DO I = 1, N
  A(I+1) = B(I) + C
  D(I) = A(I) + E
ENDDO
```

Loop Fusion (6/6)

When to Fuse Loops: Satisfying the Constraints

- The problem of minimizing the number of parallel loops using only correct *and profitable* loop fusion can be modeled as a *typed fusion* problem
    - Nearly useless description and “proof” on pp. 261–267
    - Cryptic pseudocode spanning pp. 262–263
        - Does not describe what’s happening conceptually (!)

II. Perfect Loop Nests

Loop Interchange
(Loop Skewing)

Loop Interchange, Part 1 (1/2)

Comparison with Chapter 5

- Vectorization: We moved loops to the *innermost* position

The Transformation

- Parallelization: Move dependence-free loops to the *outermost* position
  - As long as a dependence will not be introduced

```
DO I = 1, N
  DO J = 1, M
    A(I+1,J) = A(I,J)+B(I,J)
  ENDDO
ENDDO
```

```
PARALLEL DO J = 1, M
  DO I = 1, N
    A(I+1,J) = A(I,J)+B(I,J)
  ENDDO
ENDDO
```

Loop Interchange, Part 1 (2/2)

- **Effect on Dependences**
  - Recall from Chapter 5:
    1. Interchange loops ⇒ Interchange columns in direction matrix
    2. Can interchange iff all rows still have < as first non-= entry
- **When to Interchange, Part 1**
  - In a perfect loop nest, a particular loop can be parallelized at the outermost level iff its column in the direction matrix for that nest contains only "=" (Thm. 6.3)
    - Clearly, all "=" won't violate #2 above
    - But are these really the only loops? ("iff"?!)
      - If column contains >, can't move outermost by #2
      - If column contains <, can't parallelize: carries a dependence

Sequentiality Uncovers Parallelism

- If we commit to running a loop sequentially, we may be able to uncover more parallelism inside that loop
  - If we move a loop outward and sequentialize it,
    - Its column is now the first in the direction matrix
    - Remove all rows that now start with a < (deps carried by this loop)
      - Correspond to dependences that carried by the sequential loop
    - Remove its column from the direction matrix
    - Use the revised direction matrix to find parallelism inside this loop

```
DO I = 1, N
 DO J = 1, M
  DO K = 1, L
   A(I+1,J,K)    = A(I,J,K) + Q
   B(I,J,K+1)    = B(I,J,K) + R
   C(I+1,J+1,K+1)= C(I,J,K) + S
  ENDDO
 ENDDO
ENDDO
```

$$\begin{bmatrix} < & = & = \\ = & = & < \\ < & < & < \end{bmatrix}$$

Seq I

$$\begin{bmatrix} = & < \end{bmatrix}$$

```
DO I = 1, N
  PARALLEL DO J = 1, M
    DO K = 1, L
      A(I+1,J,K) = A(I,J,K) + Q
      B(I,J,K+1) = B(I,J,K) + R
      C(I+1,J+1,K+1)=C(I,J,K)+S
    ENDDO
  ENDDO
ENDDO
```

Sequentiality Uncovers Parallelism: Skewing

◆ **Effect on Dependences (Recall from §5.9)**

- Changes some = entries to <

```
DO I = 2, N+1
 DO J = 2, M+1
  DO K = 1, L
   A(I,J,K) = A(I,J-1,K) &
            + A(I-1,J,K)
   A(I,J,K+1) = B(I,J,K) &
              + A(I,J,K)
   ENDDO
 ENDDO
ENDDO
```

$$\begin{bmatrix} = & < & = \\ < & = & = \\ = & = & < \\ = & = & = \end{bmatrix}$$

loop-independent →

```
DO I = 2, N+1
 DO J = 2, M+1
  DO k = 1+I+J, L+I+J
   A(I,J,k-I-J) = A(I,J-1,k-I-J) &
                + A(I-1,J,k-I-J)
   A(I,J,k-I-J+1) = B(I,J,k-I-J) &
                  + A(I,J,k-I-J)
  ENDDO
 ENDDO
ENDDO
```

$$\begin{bmatrix} = & < & < \\ < & = & < \\ = & = & < \\ = & = & = \end{bmatrix}$$

Skew innermost loop w.r.t. the two outer loops using the substitution

`k = K + I + J`

Sequentiality Uncovers Parallelism: Skewing

- **Effect on Dependences (Recall from §5.9)**
    - Changes some = entries to <

```
DO k = 5, N+M+1
 DO I = MAX(2,k-M-L-1), MIN(N+1,k-L-2)
  DO J = MAX(2,k-I-L), MIN(M+1,k-I-1)
   A(I,J,k-I-J)   = A(I,J-1,k-I-J) &
                  + A(I-1,J,k-I-J)
   A(I,J,k-I-J+1) = B(I,J,k-I-J) &
                  + A(I,J,k-I-J)
  ENDDO
 ENDDO
ENDDO
```

Both inner loops can be parallelized!

```
DO I = 2, N+1
 DO J = 2, M+1
  DO k = 1+I+J, L+I+J
   A(I,J,k-I-J)   = A(I,J-1,k-I-J) &
                  + A(I-1,J,k-I-J)
   A(I,J,k-I-J+1) = B(I,J,k-I-J)   &
                  + A(I,J,k-I-J)
  ENDDO
 ENDDO
ENDDO
```

$$\begin{bmatrix} = & < & < \\ < & = & < \\ = & = & < \\ = & = & = \end{bmatrix}$$

Now make the innermost loop the outermost (interchange) and sequentialize it.

Both of the inner loops can then be parallelized.

Sequentiality Uncovers Parallelism: Skewing

- Skewing is useful for parallelization because it can
    - Make it possible to move a loop to the outermost position
    - Make a loop carry all the dependences originally carried by the loop w.r.t. which it was skewed
        - Running the outer loop sequentially uncovers parallelism

III. Imperfectly Nested Loops

Multilevel Loop Fusion

Multilevel Loop Fusion

The Transformation

- For imperfectly nested loops,
  - First, distribute loops maximally
  - Then try to fuse perfect nests

Multilevel Loop Fusion

When to Fuse Loop Nests: Difficulties (Example 1)

- Fusion of loop nests is actually NP-complete
- Different loop nests require different permutations
- Permutations can interfere if reassembling distributed loops
- Also memory hierarchy considerations

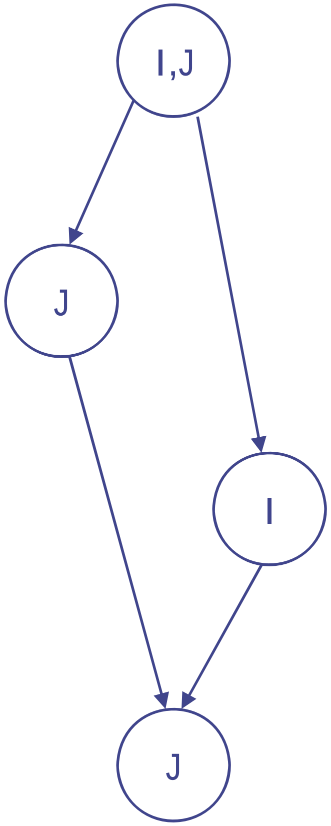
```
DO I = 1, N
  DO J = 1, M
    A(I,J+1) = A(I,J)+C
    B(I+1,J) = B(I,J)+D
  ENDDO
ENDDO
```

```
PARALLEL DO I = 1, N
  DO J = 1, M
    A(I,J+1) = A(I,J)+C
  ENDDO
ENDDO
PARALLEL DO J = 1, M
  DO I = 1, N
    B(I+1,J) = B(I,J)+D
  ENDDO
ENDDO
```

Multilevel Loop Fusion

- **When to Fuse Loop Nests: Difficulties (Example 2)**

```
DO I = 1, N  ! Can be parallel
 DO J = 1, M ! Can be parallel
  A(I,J) = A(I,J) + X
 ENDDO
ENDDO
DO I = 1, N  ! Sequential
 DO J = 1, M ! Can be parallel
  B(I+1,J) = A(I,J) + B(I,J)
 ENDDO
ENDDO
DO I = 1, N  ! Can be parallel
 DO J = 1, M ! Sequential
  C(I,J+1) = A(I,J) + C(I,J)
 ENDDO
ENDDO
DO I = 1, N  ! Sequential
 DO J = 1, M ! Can be parallel
  D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
 ENDDO
ENDDO
```



Multilevel Loop Fusion

- **When to Fuse Loop Nests: Algorithm (Heuristic)**
  - Try to parallelize individual perfect loop nests (as described earlier)
  - Then use Typed Fusion to figure out which outer loops to merge, and repeat the whole procedure for the nests inside the merged outer loops
    - The "type" of a nest has two components:
      1. The outermost loop in the resulting nest
      2. Whether this loop is sequential or parallel

```
PARALLEL DO I = 1, N
  DO J = 1, M
    A(I+1,J+1) = A(I+1,J) + C
  ENDDO
ENDDO
PARALLEL DO J = 1, M
  DO I = 1, N
    X(I,J) = A(I,J) + C
  ENDDO
ENDDO
```

← Type is (I-loop, parallel)

← Type is (J-loop, parallel)